Lecture 14

Network Flow I

14.1 Overview

In these next two lectures we are going to talk about an important algorithmic problem called the *Network Flow Problem*. Network flow is important because it can be used to express a wide variety of different kinds of problems. So, by developing good algorithms for solving network flow, we immediately will get algorithms for solving many other problems as well. In Operations Research there are entire courses devoted to network flow and its variants. Topics in today's lecture include:

- The definition of the network flow problem
- The basic Ford-Fulkerson algorithm
- The maxflow-mincut theorem
- The bipartite matching problem

14.2 The Network Flow Problem

We begin with a definition of the problem. We are given a directed graph G, a start node s, and a sink node t. Each edge e in G has an associated non-negative *capacity* $c(e)$, where for all non-edges it is implicitly assumed that the capacity is 0. For example, consider the graph in Figure 14.1 below.

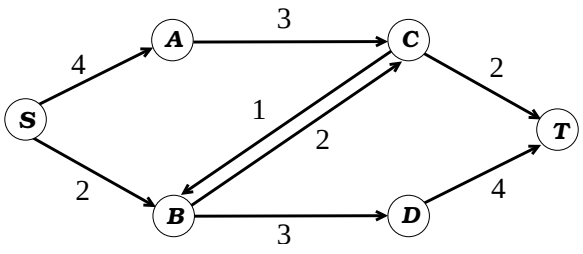


Figure 14.1: A network flow graph.

Our goal is to push as much *flow* as possible from s to t in the graph. The rules are that no edge can have flow exceeding its capacity, and for any vertex except for s and t, the flow *in* to the vertex must equal the flow *out* from the vertex. That is,

Capacity constraint: On any edge e we have $f(e) \leq c(e)$.

Flow conservation: For any vertex $v \notin \{s, t\}$, flow in equals flow out: $\sum_u f(u, v) = \sum_u f(v, u)$.

Subject to these constraints, we want to maximize the total flow into t. For instance, imagine we want to route message traffic from the source to the sink, and the capacities tell us how much bandwidth we're allowed on each edge.

E.g., in the above graph, what is the maximum flow from s to t? Answer: 5. Using "capacity[flow]" notation, the positive flow looks as in Figure 14.2. Note that the flow can split and rejoin itself.

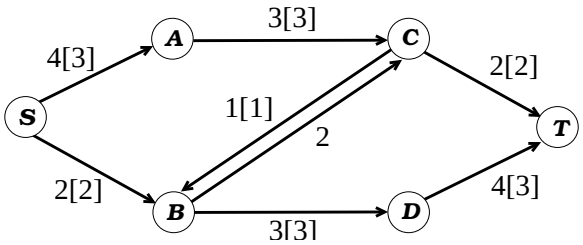


Figure 14.2: A network flow graph with positive flow shown using "capacity[flow]" notation.

How can you see that the above flow was really maximum? Notice, this flow saturates the $a \to c$ and $s \to b$ edges, and, if you remove these, you disconnect t from s. In other words, the graph has an "s-t cut" of size 5 (a set of edges of total capacity 5 such that if you remove them, this disconnects the source from the sink). The point is that any unit of flow going from s to t must take up at least 1 unit of capacity in these pipes. So, we know we're optimal.

We just argued that in general, the maximum s-t flow $\leq$ the capacity of the minimum s-t cut. An important property of flows, that we will prove as a byproduct of analyzing an algorithm for finding them, is that the maximum s-t flow is in fact *equal to* the capacity of the minimum s-t cut. This is called the *Maxflow-Mincut Theorem*. In fact, the algorithm will find a flow of some value k and a cut of capacity k, which will be proofs that both are optimal!

To describe the algorithm and analysis, it will help to be a bit more formal about a few of these quantities.

Definition 14.1 *An s-t* **cut** *is a set of edges whose removal disconnects t from s. Or, formally, a cut is a partition of the vertex set into two pieces A and B where $s \in A$ and $t \in B$. (The edges of the cut are then all edges going from A to B).*

Definition 14.2 *The* **capacity** *of a cut (A, B) is the sum of capacities of edges in the cut. Or, in the formal viewpoint, it is the sum of capacities of all edges going from A to B. (Don't include the edges from B to A.)*

Definition 14.3 *It will also be mathematically convenient for any edge (u, v) to define $f(v, u) = -f(u, v)$. This is called* **skew-symmetry**. *(We will think of flowing 1 unit on the edge from u to v as equivalently flowing -1 units on the back-edge from v to u.)*

The skew-symmetry convention makes it especially easy to add two flows together. For instance, if we have one flow with 1 unit on the edge (c, b) and another flow with 2 units on the edge (b, c), then adding them edge by edge does the right thing, resulting in a net flow of 1 unit from b to c. Also, using skew-symmetry, the total flow *out* of a node will always be the negative of the total flow *into* a node, so if we wanted we could rewrite the flow conservation condition as $\sum_u f(u, v) = 0$.

How can we find a maximum flow and prove it is correct? Here's a very natural strategy: find a path from s to t and push as much flow on it as possible. Then look at the leftover capacities (an important issue will be how exactly we define this, but we will get to it in a minute) and repeat. Continue until there is no longer any path with capacity left to push any additional flow on. Of course, we need to *prove* that this works: that we can't somehow end up at a suboptimal solution by making bad choices along the way. This approach, with the correct definition of "leftover capacity", is called the Ford-Fulkerson algorithm.

14.3 The Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm is simply the following: while there exists an $s \to t$ path P of positive *residual capacity* (defined below), push the maximum possible flow along P. By the way, these paths P are called *augmenting paths*, because you use them to augment the existing flow.

Residual capacity is just the capacity left over given the existing flow, where we will use skew-symmetry to capture the notion that if we push f units of flow on an edge (u, v), this *increases* our ability to push flow on the back-edge (v, u) by f.

Definition 14.4 *Given a flow f in graph G, the* **residual capacity** *$c_f(u, v)$ is defined as $c_f(u, v) = c(u, v) - f(u, v)$, where recall that by skew-symmetry we have $f(v, u) = -f(u, v)$.*

For example, given the flow in Figure 14.2, the edge (s, a) has residual capacity 1. The back-edge (a, s) has residual capacity 3, because its original capacity was 0 and we have $f(a, s) = -3$.

Definition 14.5 *Given a flow f in graph G, the* **residual graph** *G_f is the directed graph with all edges of positive residual capacity, each one labeled by its residual capacity. Note: this may include back-edges of the original graph G.*

Let's do an example. Consider the graph in Figure 14.1 and suppose we push two units of flow on the path $s \to b \to c \to t$. We then end up with the following residual graph:

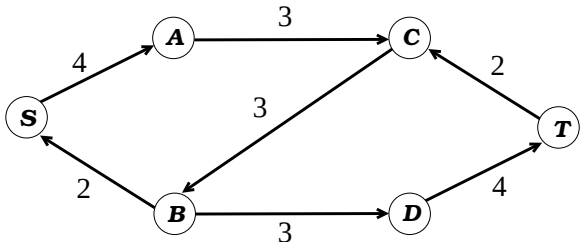


Figure 14.3: Residual graph resulting from pushing 2 units of flow along the path s-b-c-t in the graph in Figure 14.1.

If we continue running Ford-Fulkerson, we see that in this graph the only path we can use to augment the existing flow is the path $s \to a \to c \to b \to d \to t$. Pushing the maximum 3 units on this path we then get the next residual graph, shown in Figure 14.4. At this point there is no

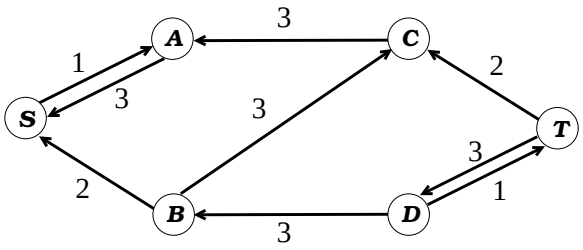


Figure 14.4: Residual graph resulting from pushing 3 units of flow along the path s-a-c-b-d-t in the graph in Figure 14.3.

longer a path from s to t so we are done.

We can think of Ford-Fulkerson as at each step finding a new flow (along the augmenting path) and adding it to the existing flow, where by *adding* two flows we mean adding them edge by edge. Notice that the sum of two flows continues to satisfy flow-conservation and skew-symmetry. The definition of residual capacity ensures that the flow found by Ford-Fulkerson is *legal* (doesn't exceed the capacity constraints in the original graph). We now need to prove that in fact it is *maximum*. We'll worry about the number of iterations it takes and how to improve that later.

Note that one nice property of the residual graph is that it means that at each step we are left with same type of problem we started with. So, to implement Ford-Fulkerson, we can use any black-box path-finding method (e.g., DFS).

Theorem 14.1 *The Ford-Fulkerson algorithm finds a maximum flow.*

Proof: Let's look at the final residual graph. This graph must have s and t disconnected by definition of the algorithm. Let A be the component containing s and B be the rest. Let c be the capacity of the (A, B) cut in the *original* graph — so we know we can't do better than c.

The claim is that we in fact *did* find a flow of value c (which therefore implies it is maximum). Here's why: let's look at what happens to the residual capacity of the (A, B) cut after each iteration of the algorithm. Say in some iteration we found a path with k units of flow. Then, even if the path zig-zagged between A and B, every time we went from A to B we added k to the flow from A to B and subtracted k from the residual capacity of the (A, B) cut, and every time we went from B to A we took away k from this flow and added k to the residual capacity of the cut[1]; moreover, we must have gone from A to B *exactly* one more time than we went from B to A. So, the residual capacity of this cut went down by exactly k. So, the drop in capacity is equal to the increase in flow. Since at the end the residual capacity is zero (remember how we defined A and B) this means the total flow is equal to c.

So, we've found a flow of value *equal* to the capacity of this cut. We know we can't do better, so this must be a max flow, and (A, B) must be a minimum cut. ■

[1]This is where we use the fact that if we flow k units on the edge (u, v), then in addition to reducing the residual capacity of the (u, v) edge by k we also *add* k to the residual capacity of the back-edge (v, u).

Notice that in the above argument we actually proved the nonobvious *maxflow-mincut* theorem:

Theorem 14.2 *In any graph G, for any two vertices s and t, the maximum flow from s to t equals the capacity of the minimum (s,t)-cut.*

We have also proven the *integral-flow theorem*: if all capacities are integers, then there is a maximum flow in which all flows are integers. This seems obvious, but you'll use it to show something that's not at all obvious on homework 5!

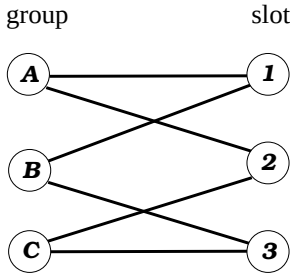
In terms of running time, if all capacities are integers and the maximum flow is F, then the algorithm makes at most F iterations (since each iteration pushes at least one more unit of flow from s to t). We can implement each iteration in time $O(m+n)$ using DFS. So we get the following result.

Theorem 14.3 *If the given graph G has integer capacities, Ford-Fulkerson runs in time $O(F(m+n))$ where F is the value of the maximum s-t flow.*

In the next lecture we will look at methods for reducing the number of iterations the algorithm can take. For now, let's see how we can use an algorithm for the max flow problem to solve other problems as well: that is, how we can *reduce* other problems to the one we now know how to solve.

14.4 Bipartite Matching

Say we wanted to be more sophisticated about assigning groups to homework presentation slots. We could ask each group to list the slots acceptable to them, and then write this as a bipartite graph by drawing an edge between a group and a slot if that slot is acceptable to that group. For example:



This is an example of a **bipartite graph**: a graph with two sides L and R such that all edges go between L and R. A **matching** is a set of edges with no endpoints in common. What we want here in assigning groups to time slots is a **perfect matching**: a matching that connects every point in L with a point in R. For example, what is a perfect matching in the bipartite graph above?

More generally (say there is no perfect matching) we want a **maximum matching**: a matching with the maximum possible number of edges. We can solve this as follows:

Bipartite Matching:

1. Set up a fake "start" node s connected to all vertices in L. Connect all vertices in R to a fake "sink" node T. Orient all edges left-to-right and give each a capacity of 1.
2. Find a max flow from s to t using Ford-Fulkerson.

3. Output the edges between L and R containing nonzero flow as the desired matching.

This finds a legal matching because edges from R to t have capacity 1, so the flow can't use two edges *into* the same node, and similarly the edges from s to L have capacity 1, so you can't have flow on two edges *leaving* the same node in L. It's a *maximum* matching because any matching gives you a flow of the same value: just connect s to the heads of those edges and connect the tails of those edges to t. (So if there was a better matching, we wouldn't be at a maximum flow).

What about the number of iterations of path-finding? This is at most the number of edges in the matching since each augmenting path gives us one new edge.

Let's run the algorithm on the above example. Notice a neat fact: say we start by matching A to 1 and C to 3. These are bad choices, but the augmenting path *automatically* undoes them as it improves the flow!

Matchings come up in many different problems like matching up suppliers to customers, or cell-phones to cell-stations when you have overlapping cells. They are also a basic part of other algorithmic problems.